Gamma Random Number Generation on GPUs using CUDA

Johan Ericsson
June 19, 2024 — KTH Royal Institute of Technology

Table of contents

Introduction

Motivation

- Stochastic simulation techniques play an important role in modern society with a diverse range of applications:

Motivation

- Stochastic simulation techniques play an important role in modern society with a diverse range of applications:

  1. telecommunication systems
  2. nuclear physics
  3. weather forecasting and climate modeling

Motivation

- Stochastic simulation techniques play an important role in modern society with a diverse range of applications:
    1. telecommunication systems
    2. nuclear physics
    3. weather forecasting and climate modeling
- Monte Carlo Simulations has been used since the Manhattan Project.

Motivation

- Stochastic simulation techniques play an important role in modern society with a diverse range of applications:
  1. telecommunication systems
  2. nuclear physics
  3. weather forecasting and climate modeling
- Monte Carlo Simulations has been used since the Manhattan Project.
- Require that we can simulate random variables efficiently!

Changing Compute Landscape

- Graphical processing units (GPUs) and other accelerators are becoming more important.

Changing Compute Landscape

- Graphical processing units (GPUs) and other accelerators are becoming more important.
- The computer architecture of GPUs differ from that of classical central processing units (CPUs).

Changing Compute Landscape

- Graphical processing units (GPUs) and other accelerators are becoming more important.
- The computer architecture of GPUs differ from that of classical central processing units (CPUs).
- Algorithms that perform well on CPUs may not perform well on GPUs and vice versa.

Changing Compute Landscape

- Graphical processing units (GPUs) and other accelerators are becoming more important.
- The computer architecture of GPUs differ from that of classical central processing units (CPUs).
- Algorithms that perform well on CPUs may not perform well on GPUs and vice versa.
- Challenges when implementing code that require random numbers on GPUs:
  1. Poor library support for complex distributions (e.g. gamma)
  2. Much of the existing literature is focused on CPUs and not GPUs.

In this work we

- We present the first comparison of the performance of gamma random number generation algorithms on GPUs.

In this work we

- We present the first comparison of the performance of gamma random number generation algorithms on GPUs.
- We describe the implementation and design of efficient random number generation kernels on GPUs.

In this work we

- We present the first comparison of the performance of gamma random number generation algorithms on GPUs.
- We describe the implementation and design of efficient random number generation kernels on GPUs.
- Our results show that a 1000× speedup can be achieved when generating gamma random numbers on a consumer grade GPU compared to on a CPU (single thread).

Background

Gamma Distribution

Let $\alpha, \beta > 0$ be real numbers, then the *gamma distribution* $\Gamma(\alpha, \beta)$ has p.d.f.

$$f(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-x/\beta}, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

The parameters are called: *shape* (α) and *scale* (β).

Gamma Distribution

Let $\alpha, \beta > 0$ be real numbers, then the *gamma distribution* $\Gamma(\alpha, \beta)$ has p.d.f.

$$f(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^{\alpha}} x^{\alpha-1} e^{-x/\beta}, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

The parameters are called: *shape* (α) and *scale* (β).

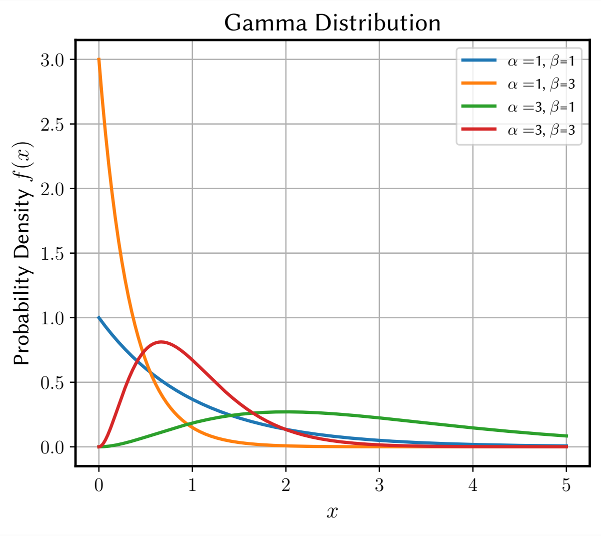


Figure: Gamma Distribution for different shape and scale parameters.

Gamma Distribution

1. If $X \sim \Gamma(1, \beta)$, then $X \sim \text{Exp}(\lambda)$ with $\lambda = \frac{1}{\beta}$.

Gamma Distribution

1. If $X \sim \Gamma(1, \beta)$, then $X \sim \text{Exp}(\lambda)$ with $\lambda = \frac{1}{\beta}$.
2. If $X_i \sim \Gamma(\alpha_i, \beta)$, then

$$cX_1 + \cdots + cX_n \in \Gamma(\alpha_1 + \cdots + \alpha_n, c\beta).$$

Gamma Distribution

1. If $X \sim \Gamma(1, \beta)$, then $X \sim \text{Exp}(\lambda)$ with $\lambda = \frac{1}{\beta}$.
2. If $X_i \sim \Gamma(\alpha_i, \beta)$, then
$$cX_1 + \cdots + cX_n \in \Gamma(\alpha_1 + \cdots + \alpha_n, c\beta).$$
3. If $Y \sim \Gamma(\alpha + 1, 1)$ for some $\alpha > 0$ and $U \sim U(0, 1)$ be independent random variables, then
$$X = YU^{1/\alpha} \sim \Gamma(\alpha, 1).$$

Item 2 and 3 above implies that $\Gamma(\alpha, 1)$ $(\alpha > 1)$ variates can be cheaply transformed to $\Gamma(\alpha, \beta)$ variates for arbitrary α, β. Hence, our focus is on $\Gamma(\alpha, 1)$ generators for $\alpha > 1$.

Random Number Generation

- There are two types of random number generators (RNGs) uniform and non-uniform.

Random Number Generation

- There are two types of random number generators (RNGs) uniform and non-uniform.

- Uniform RNGs produce uniform $U(0, 1)$ random samples (random bits).

Random Number Generation

- There are two types of random number generators (RNGs) uniform and non-uniform.
- Uniform RNGs produce uniform $U(0, 1)$ random samples (random bits).
- Non-uniform RNGs use uniform RNGs for randomness combined with mathematical transforms to generate samples from other distributions.

Random Number Generation

- There are two types of random number generators (RNGs) uniform and non-uniform.
- Uniform RNGs produce uniform $U(0,1)$ random samples (random bits).
- Non-uniform RNGs use uniform RNGs for randomness combined with mathematical transforms to generate samples from other distributions.
- Only one method is used for gamma generation: rejection sampling.

GPU Architecture

- At the highest level a GPU consists of several *streaming multiprocessors (SMs)*.
- The SMs have a *single instruction, multiple threads (SIMT)* design: many compute cores each have their own registers and but are collected in groups which share the same instruction control unit.

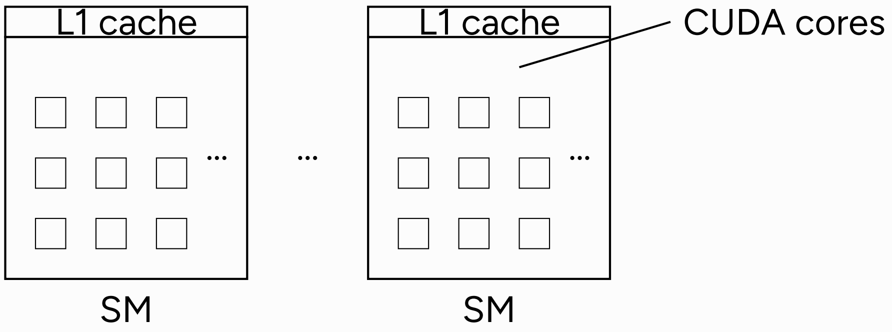


Figure: Illustration of GPU architecture showing SMs, CUDA cores, and L1 cache.

Rejection Sampling

Algorithm 1: Rejection Sampling

Data: Desired distribution f, proposal distribution g, constant M

Result: Sample from distribution X

Input: Initialize $accepted \leftarrow$ false

```
while not accept do
    Sample y ~ Y
    Sample u ~ U(0, 1)
    if u < f(y) / (M·g(y)) then
        accept ← true
    end
    return y
end
```

Rejection Sampling on GPUs

Large difference between CPUs and GPUs when it comes to efficient rejection samling:

Rejection Sampling on GPUs

Large difference between CPUs and GPUs when it comes to efficient rejection samling:

- Warps and SIMT architecture leads to different rejection distributions (see [8]).

Rejection Sampling on GPUs

Large difference between CPUs and GPUs when it comes to efficient rejection samling:

- Warps and SIMT architecture leads to different rejection distributions (see [8]).

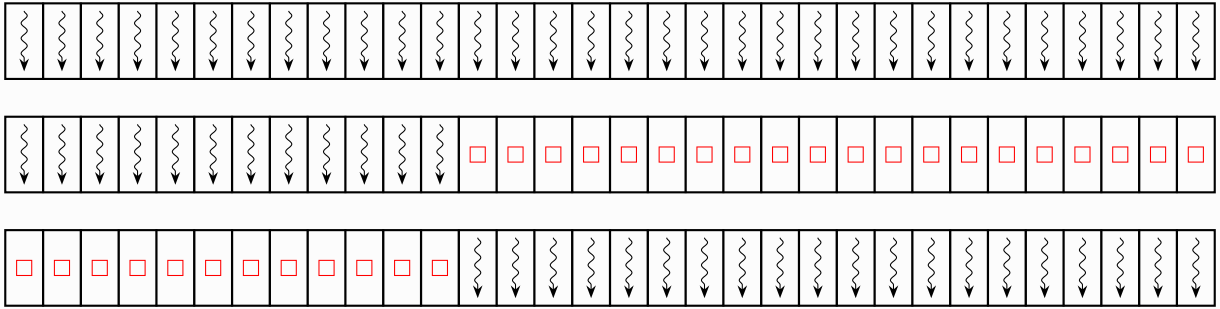


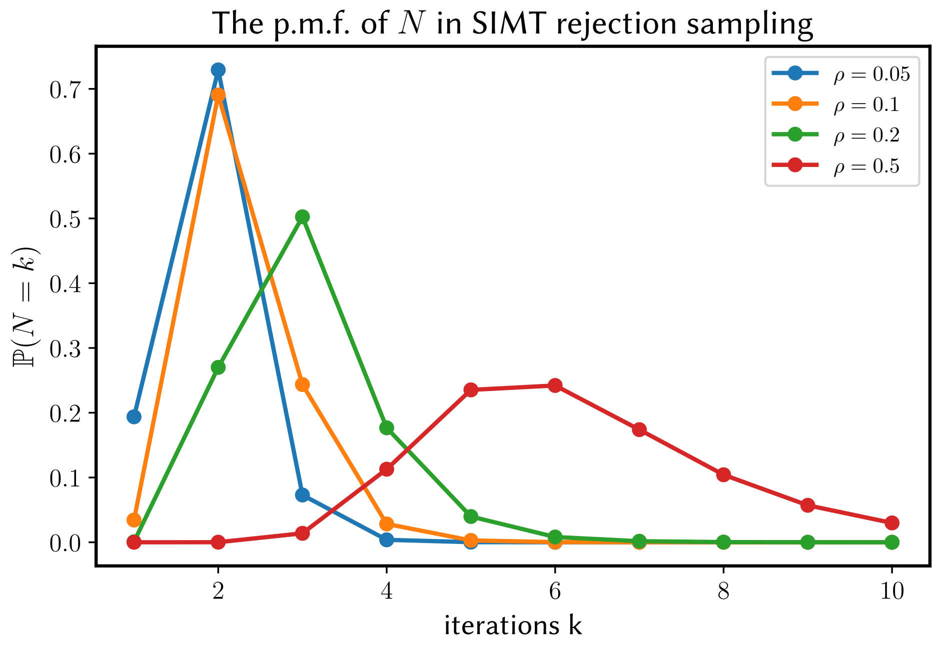
Figure: Visualization of warp divergence. The arrow indicates that the thread is doing work and the red square indicates that the thread is idle.

Analytical formula for the probability

$$P(N = k) = (1 - \rho^n)^t - \left(1 - \rho^{n-1}\right)^t,$$

where:

- N - number iterations until accept
- t - warp size (32 for NVIDIA GPUs)
- ρ - rejection probability



Methods

Implementation of Kernels

- We selected and implemented a selection of existing gamma generation algorithm in CUDA as `__device__` kernels

Implementation of Kernels

- We selected and implemented a selection of existing gamma generation algorithm in CUDA as `__device__` kernels

- Benchmarking class was written using C++ templates with the device kernel passed as a template parameter

Implementation of Kernels

- We selected and implemented a selection of existing gamma generation algorithm in CUDA as `__device__` kernels
- Benchmarking class was written using C++ templates with the device kernel passed as a template parameter
- Flexibility in selecting kernel to benchmark without the overhead of any runtime dispatch.

Implementation of Kernels

- We selected and implemented a selection of existing gamma generation algorithm in CUDA as `__device__` kernels
- Benchmarking class was written using C++ templates with the device kernel passed as a template parameter
- Flexibility in selecting kernel to benchmark without the overhead of any runtime dispatch.
- The gamma generation benchmark class use persistent threads (PT) [2], [6].

Selected Kernels

We selected 5 kernels that we believe can be be efficiently implemented on the GPU:

- Best (XG) [3]
- Cheng (GA) [4]
- Cheng-Feast (GMK3) [5]
- Ahrens-Dieter (GC) [1]
- Marsaglia-Tsang [7]

Selected Kernels

We selected 5 kernels that we believe can be be efficiently implemented on the GPU:

- Best (XG) [3]
- Cheng (GA) [4]
- Cheng-Feast (GMK3) [5]
- Ahrens-Dieter (GC) [1]
- Marsaglia-Tsang [7]

The first four are published in the 1970s and Marsaglia-Tsang in 2000. Their measurements are on ~ 50 year old computer hardware!

Verification of implementations

- Used a Kolmogorov-Smirnov test (KS-test) to verify that the implementations are correct and generate gamma distributed output.

Verification of implementations

- Used a Kolmogorov-Smirnov test (KS-test) to verify that the implementations are correct and generate gamma distributed output.

- We know mathematically that the output should be gamma distributed.

Verification of implementations

- Used a Kolmogorov-Smirnov test (KS-test) to verify that the implementations are correct and generate gamma distributed output.

- We know mathematically that the output should be gamma distributed.

- Output quality depends on the uniform RNG. CUDAs default uniform RNG was used: `CURAND_RNG_PSEUDO_XORWOW`

Measurements

- Measured the execution times for each gamma kernel when generating samples of single precision floating-point numbers.

Measurements

- Measured the execution times for each gamma kernel when generating samples of single precision floating-point numbers.

- Varying sample sizes between $2^{22} \approx 4 \times 10^6$ and $2^{28} \approx 2.68 \times 10^8$.

Measurements

- Measured the execution times for each gamma kernel when generating samples of single precision floating-point numbers.

- Varying sample sizes between $2^{22} \approx 4 \times 10^6$ and $2^{28} \approx 2.68 \times 10^8$.

- The measurements were done for four different values of $\alpha = 1.0001, 2, 4, 10$

Measurements

- Measured the execution times for each gamma kernel when generating samples of single precision floating-point numbers.
- Varying sample sizes between $2^{22} \approx 4 \times 10^6$ and $2^{28} \approx 2.68 \times 10^8$.
- The measurements were done for four different values of $\alpha = 1.0001, 2, 4, 10$
- Warmup iteration + 10 measurements, means are reported in figures with variance errorbars.

Experimental Setup

- Linux host running Ubuntu 22.04.3 LTS with linux kernel version 5.15.0-58-generic.
- AMD Ryzen 9 5950X 16-Core CPU with clock frequency 3.4GHz and memory listed in table 1.

L1d cache	512 KiB
L1i cache	512 KiB
L2 cache	8 MiB
L3 cache	64 MiB
RAM	32 GiB (2x16 GiB)
SSD	1TB

Table: Cache sizes for the AMD 5950X CPU used for benchmarking and installed memory sizes.

Experimental Setup

- NVIDIA GeForce RTX 4070 GPU

GPU Architecture	Ada Lovelace
CUDA Cores	5888
Clock Speed	1.92 GHz
RAM	12 GiB
Memory Interface	192-bit
Memory Bandwidth	504.2 GB/s
L1 Cache Size	192 KiB per SM
L2 Cache Size	36 MiB

Table: Key stats for the NVIDIA GeForce RTX 4070 GPU used for measurements.

Results

Verification of Output

Algorithm	$\alpha = 1.0001$		$\alpha = 2.0$		$\alpha = 10.0$	
	D_n	*p-value*	D_n	*p-value*	D_n	*p-value*
Cheng-Feast (GKM3)	0.0012	0.11	0.00094	0.34	0.0015	0.018
Marsaglia-Tsang	0.00059	0.88	0.00069	0.72	0.00072	0.67
Cheng (GA)	0.00067	0.76	0.00052	0.95	0.00074	0.64
Best (XG)	0.00069	0.73	0.00059	0.87	0.00062	0.84
Ahrens-Dieter (GC)	0.00063	0.83	0.00059	0.88	0.00064	0.80

Table: KS-test results of the algorithms for selected values of α.

Verification of Output

Algorithm	α = 1.0001		α = 2.0		α = 10.0	
	D_n	p-value	D_n	p-value	D_n	p-value
Cheng-Feast (GKM3)	0.0012	0.11	0.00094	0.34	0.0015	0.018
Marsaglia-Tsang	0.00059	0.88	0.00069	0.72	0.00072	0.67
Cheng (GA)	0.00067	0.76	0.00052	0.95	0.00074	0.64
Best (XG)	0.00069	0.73	0.00059	0.87	0.00062	0.84
Ahrens-Dieter (GC)	0.00063	0.83	0.00059	0.88	0.00064	0.80

Table: KS-test results of the algorithms for selected values of α.

The p-values suggest that all algorithms produce gamma distributed output, except Cheng-Feast (GKM3) for high α which is much worse than the other algorithms.

Verification of Output

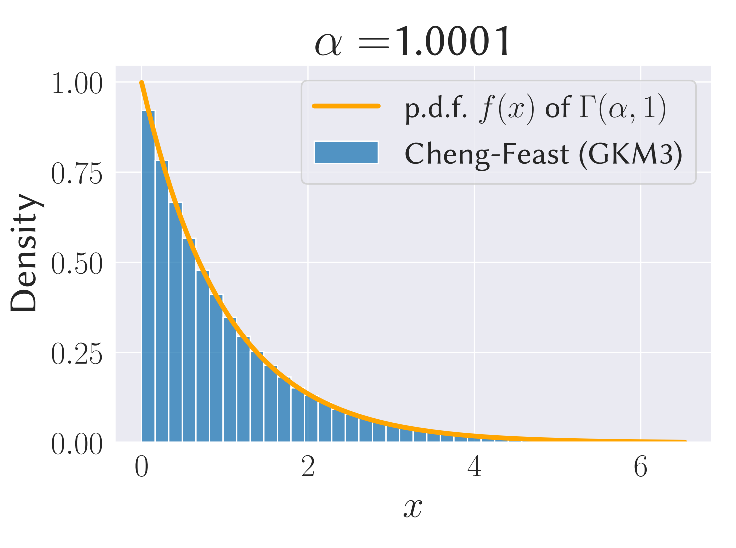


Figure: Histogram of output of Cheng-Feast (GKM3) 10^6 samples.

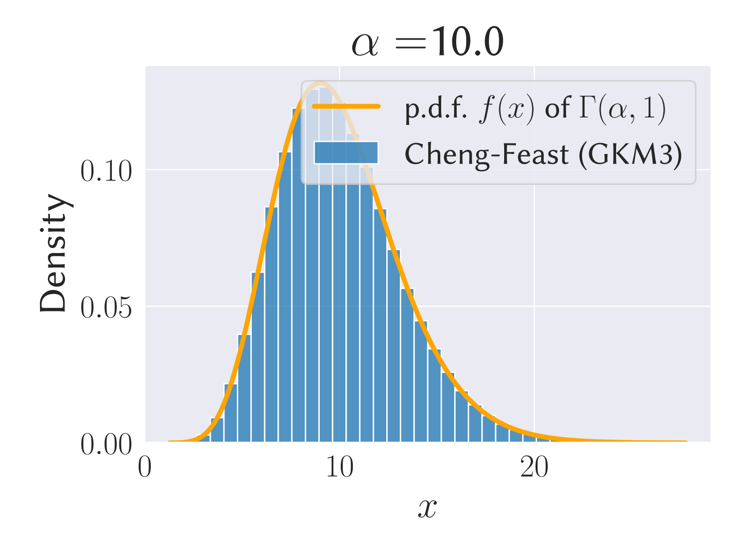


Figure: Histogram of output of Cheng-Feast (GKM3) 10^6 samples.

Execution times α = 1.0001

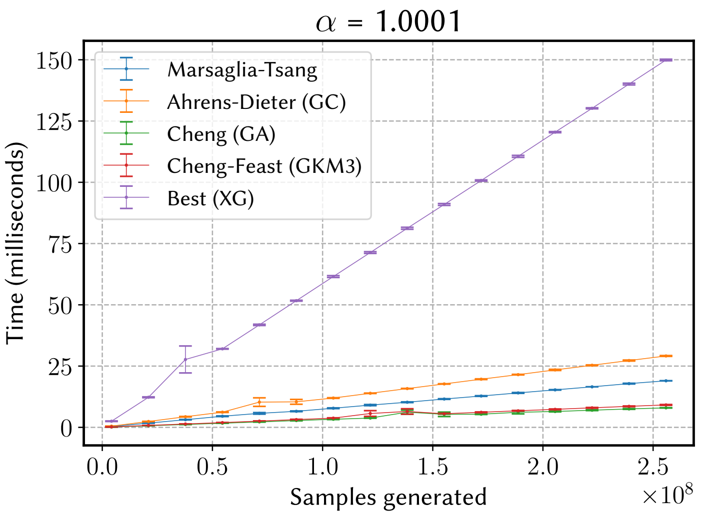


Figure: Measured execution times for α = 1.0001.

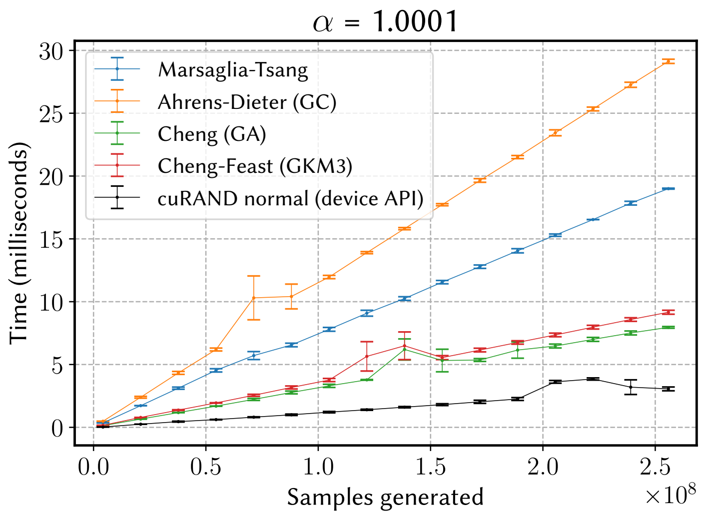


Figure: Measured execution times for the best kernels α = 1.0001 and with cuRAND normal.

Execution times α = 10

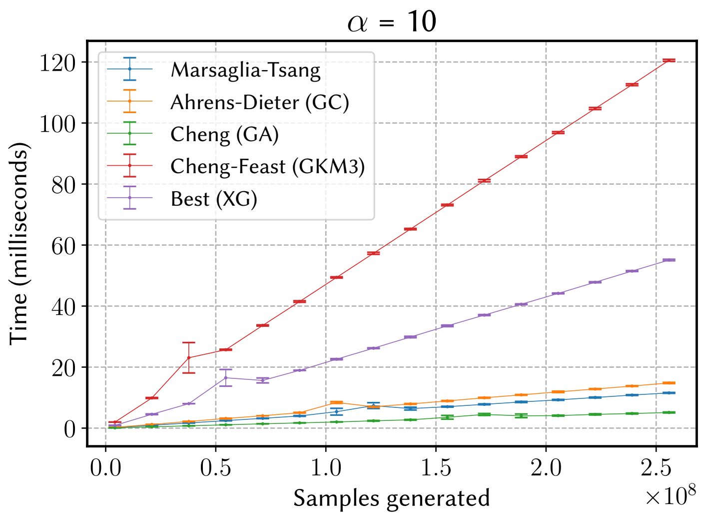


Figure: Measured execution times for α = 10.

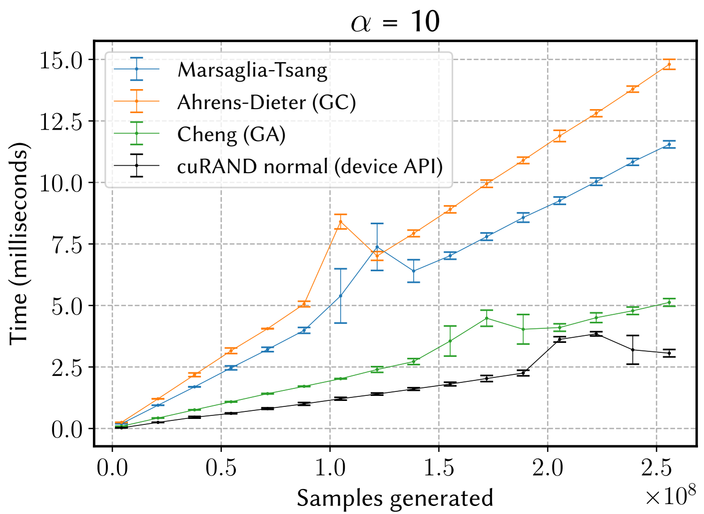


Figure: Measured execution times for the best kernels α = 10 and with cuRAND normal.

Speedups

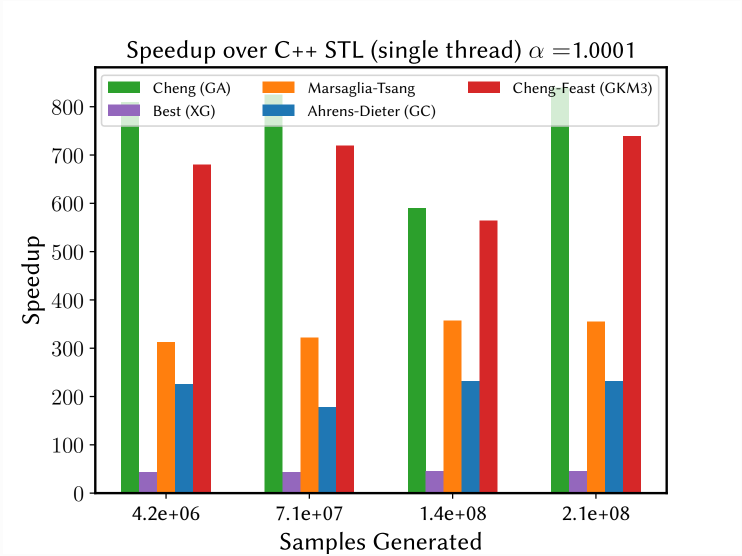


Figure: Speedup compared to CPU single thread (C++ STL) for $\alpha = 1.0001$.

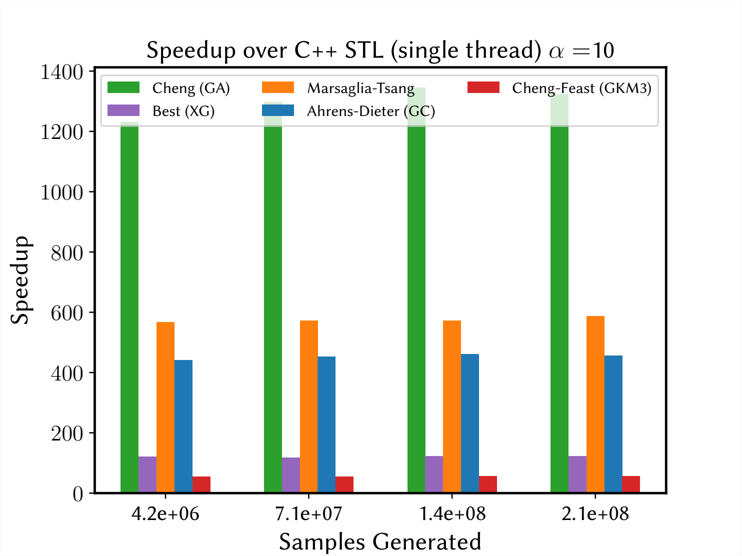


Figure: Speedup compared to CPU single thread (C++ STL) for $\alpha = 10$.

Conclusions

Conclusions

- It is possible to efficiently generate gamma random numbers on GPUs.

Conclusions

- It is possible to efficiently generate gamma random numbers on GPUs.
- The best algorithm is Cheng (GA) [4] which perform very well on the GPU across all $\alpha > 1$.

Conclusions

- It is possible to efficiently generate gamma random numbers on GPUs.
- The best algorithm is Cheng (GA) [4] which perform very well on the GPU across all $\alpha > 1$.
    - (Not often mentioned in the literature, which is focused on CPUs).

Conclusions

- It is possible to efficiently generate gamma random numbers on GPUs.
- The best algorithm is Cheng (GA) [4] which perform very well on the GPU across all $\alpha > 1$.
  - (Not often mentioned in the literature, which is focused on CPUs).
  - Achieves > $1000\times$ speedup compared to CPU for $\alpha > 2$.

Conclusions

- It is possible to efficiently generate gamma random numbers on GPUs.
- The best algorithm is Cheng (GA) [4] which perform very well on the GPU across all $\alpha > 1$.
  - (Not often mentioned in the literature, which is focused on CPUs).
  - Achieves > $1000\times$ speedup compared to CPU for $\alpha > 2$.
  - Easy to implement ~ 25 lines of code.

Conclusions

- It is possible to efficiently generate gamma random numbers on GPUs.
- The best algorithm is Cheng (GA) [4] which perform very well on the GPU across all $\alpha > 1$.
  - (Not often mentioned in the literature, which is focused on CPUs).
  - Achieves > $1000\times$ speedup compared to CPU for $\alpha > 2$.
  - Easy to implement ~ 25 lines of code.
- Shows that rejection sampling does not have to be ”bad” on GPUs.

Future Work

- The best algorithms for generating random numbers from other complex distributions on GPUs are not known.

Future Work

- The best algorithms for generating random numbers from other complex distributions on GPUs are not known.

- A natural question is whether the performance comparison on CPUs are still valid?

Future Work

- The best algorithms for generating random numbers from other complex distributions on GPUs are not known.

- A natural question is whether the performance comparison on CPUs are still valid?

- The same work can be done for modern CPUs.

Questions

Feel free to ask questions!

References I

[1] Joachim H. Ahrens and Ulrich Dieter. “Computer methods for sampling from gamma, beta, poisson and bionomial distributions”. In: *Computing* 12.3 (1974), pp. 223–246. DOI: 10.1007/BF02293108. URL: https://doi.org/10.1007/BF02293108.

[2] Timo Aila and Samuli Laine. “Understanding the efficiency of ray traversal on GPUs”. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG ’09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149. ISBN: 9781605586038. DOI: 10.1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792.

References II

[3] D. J. Best. “Letters to the Editors”. eng. In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 27.2 (1978), pp. 181–182. ISSN: 0035-9254. DOI: 10.1111/j.1467-9876.1978.tb01041.x. URL: https://doi.org/10.1111/j.1467-9876.1978.tb01041.x.

[4] R. C. H. Cheng. “The Generation of Gamma Variables with Non-Integral Shape Parameter”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 26.1 (1977), pp. 71–75. URL: http://www.jstor.org/stable/2346871 (visited on 05/17/2024).

References III

[5] R. C. H. Cheng and G. M. Feast. “Some Simple Gamma Variate Generators”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.3 (1979), pp. 290–295. DOI: 10.2307/2347200. URL: https://doi.org/10.2307/2347200.

[6] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. “A study of Persistent Threads style GPU programming for GPGPU workloads”. In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–14. DOI: 10.1109/InPar.2012.6339596.

[7] George Marsaglia and Wai Wan Tsang. “A simple method for generating gamma variables”. In: *ACM Trans. Math. Softw.* 26.3 (2000), pp. 363–372. DOI: 10.1145/358407.358414. URL: https://doi.org/10.1145/358407.358414.

References IV

[8] Gavin Ridley and Benoit Forget. “A simple method for rejection sampling efficiency improvement on SIMT architectures”. In: *Stat. Comput.* 31.3 (2021), p. 30. DOI: 10.1007/S11222-021-10003-Z. URL: https://doi.org/10.1007/s11222-021-10003-z.

KTH

VETENSKAP
OCH KONST